Peer-based Automatic Configuration of Pervasive Applications

MARCUS HANDTE, CHRISTIAN BECKER AND KURT ROTHERMEL
Institute of Parallel and Distributed Systems, Universität Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany
Email: firstname.lastname@informatik.uni-stuttgart.de

Received: October 24 2005

***Abstract*— Pervasive computing envisions seamless support for user tasks through cooperating devices that are present in an environment. Fluctuating availability of devices, induced by mobility and failures, requires mechanisms and algorithms that allow applications to adapt to their ever-changing execution environments without user intervention. To ease the development of adaptive applications, Becker et al. [3] have proposed the peer-based component system PCOM. This system provides fundamental mechanisms to support the automated composition of applications at runtime. In this article, we discuss the requirements on algorithms that enable automatic configuration of pervasive applications. Furthermore, we show how finding a configuration can be interpreted as Distributed Constraint Satisfaction Problem. Based on this, we present an algorithm that is capable of finding an application configuration in the presence of strictly limited resources. To show the feasibility of this algorithm, we present an evaluation based on simulations and real-world measurements and we compare the results with a simple greedy approximation.**

***Index Terms*— Pervasive Computing, Configuration, Components, Resources, Constraint Satisfaction**

I. Introduction

Pervasive Computing utilizes devices that cooperatively execute distributed applications in order to provide distraction-free support for complex user tasks. In essence, pervasive applications can be seen as compositions of functionality provided by devices in the physical environment of their users. The interaction between applications and users is unobtrusive since many devices become invisible through their integration in everyday objects. The devices encountered in such environments are heterogeneous, ranging from resource-limited specialized systems up to powerful general purpose computers. Due to wireless communication, many devices can be mobile. Hence, the available functionality is continuously fluctuating.

Both, the heterogeneity and the dynamics of the environment increase the complexity that developers, administrators, and users face when they are building, operating, or using applications. While it is possible to shift the responsibilities and thus, the arising complexities, between these parties, e.g. administration requires more fine-tuning but usage becomes simpler, a pure shift is not enough to cope with the complexities.

In response, a number of research projects are focusing on the development of abstractions that enable the automation of various aspects of pervasive systems. One of these aspects is the automatic composition of applications at runtime. This automation is a major rationale behind many pervasive infrastructures, e.g. GAIA [17], AURA [9], and the component system PCOM [3]. At the present time, automated application composition is receiving attention in other research areas as well, e.g. the multimedia [11] and the web services community [16].

In PCOM, the configuration of a component-based application, i.e. the composition of components that constitute an application is automatically determined at runtime. If the resources required by components are limited, finding a single configuration that meets all requirements is an NP-complete problem.

In this article, we derive the requirements on peer-based algorithms that enable automatic configuration of pervasive applications. Furthermore, we propose an approach towards automatic configuration of PCOM applications based on existing work in the field of Distributed Constraint Satisfaction [22]. In contrast to the previously introduced greedy heuristic [3], the proposed approach is complete. Despite the exponential runtime, our evaluation suggests that a) the approach can be applied to many real-world problems and b) even with limited runtime it can easily outperform the greedy heuristic.

The remainder of this article is structured as follows. The next section introduces the underlying system model and presents the relevant concepts of PCOM. Section III describes the configuration process and derives the requirements for its automation. The overall approach and the interpretation of automatic configuration as Distributed Constraint Satisfaction Problem is motivated and detailed in Section IV. Section V provides an overview of the resulting algorithm that is evaluated in Section VI. Finally, Section VII describes related work and Section VIII concludes the article.

II. System Model

As presented in [3] and [4], our work focuses on peer-based pervasive systems. In such systems, devices within communication range connect to each other on-the-fly using wireless communication technology, e.g. Bluetooth or WLAN. Devices offer their functionality and cooperate with other

devices in the vicinity in order to execute applications. As a result, applications are composed of functionalities provided by different devices. Due to user mobility, the availability of functionalities is continuously fluctuating.

In contrast to smart environments like GAIA [17], AURA [9] and iROS [15], peer-based systems do not rely on the presence of a centralized coordinating entity. Thus, these systems do not require any central infrastructure that provides basic services which has the potential to broaden their applicability. As an example consider a group of persons that is cooperatively working with PDAs on a business trip. In this scenario, relying on a central coordinating entity, e.g. a fixed server, might prohibit cooperation. Another benefit of peer-based systems is the fact that due to the lack of infrastructure, deployment and maintenance is practically free.

To spontaneously form a peer-based system, each device needs to be equipped with local representatives of all basic services such as device discovery or naming services. These services must then federate themselves seamlessly which might have a negative impact on their overall efficiency. This drawback, however, can be greatly reduced by applying adequate protocols that automatically elect coordinating entities whenever a group of devices stays within communication range for a longer period of time [19].

A. PCOM

To ease the development of applications for peer-based pervasive systems, Becker et al. have proposed the light-weight component system PCOM. In the following, we sketch the relevant concepts of this system. A more detailed description can be found in [3].

In PCOM, applications are trees of component instances that are typically distributed across a set of devices. Each component is atomic with respect to distribution, i.e. it runs exactly on one device. Each component instance that is part of an application provides a certain functionality to its parent instance. To do this, each instance relies on the functionalities provided by its children and it may additionally require resources provided by the executing device.

To enable automatic configuration, each component instance explicitly declares its dependencies towards other components and resources as well as its provided functionality within a so-called contract. In [5], Beugnard et al. identify four levels of component contracts: syntactic, behavioral, synchronization and QoS contracts. At the present time, PCOM component contracts model the syntactic level, i.e. interfaces, and they can be enriched with parameters to model QoS attributes as well. The communication between component instances is bi-directional. A parent instance may invoke methods on its children and a child may emit events to its parent instance. As a result, the syntactic requirements and provisions are expressed in terms of offered and required interfaces and events. All elements and attributes of PCOM contracts allow automated matching at runtime. Thus, the system can automatically determine whether a certain dependency or resource requirement can be fulfilled by a certain offer.

A component container that is running on each device is responsible for managing the life-cycle of its local components. Conceptionally, the life-cylce of each instance consists of a started and a stopped state. The container guarantees that as long as a component instance is in the started state, its dependencies are resolved, i.e. the container has bound a suitable instance to each declared dependency, and all required resources are available. As a result, the life-cycle of the root component instance, the so-called application anchor, defines the overall life-cycle of an application. Whenever a component is started, the component container must resolve all dependencies recursively according to the corresponding contracts. Whenever the instance is stopped, the corresponding component instances will be stopped recursively by the container.

Clearly, the process of starting an application would be very resource intensive if the container would have to instantiate components in order to determine whether a certain tree can be started. Therefore, each component implementation additionally provides a so-called component factory. The factory enables the container to create contracts that match a certain requirement without actually instanciating the component. This way, the component container can test whether a certain tree can be started without actually starting it.

Conceptionally, the number of instances that can be created from a component is is only limited by the available resources. It is important to note that these resources can be strictly limited, e.g. to model exclusive resources like input devices or LEDs on some smart object. Together with the fact that different components might have overlapping resource requirements, this strict limitation on resources leads to the problem that although two subtrees of an application might be startable independently, they might not be startable simultaneously.

III. Automatic Configuration

Automatic configuration denotes the task of automatically determining a composition of components that can be instantiated simultaneously as application. Such a composition is subject to two classes of constraints. The first class are structural constraints. They describe what constitutes a valid composition in terms of functionalities. The second class are resource constraints. They are a result of the limited resources.

Structural constraints can be either specified in advance, e.g. as an architectural model expressed in some description language, or they can be individually associated with components, e.g. as recursively specified dependencies contained in contracts. If an architectural model is available, the configuration must ensure the availability of a matching instance for each modeled component. If structural constraints are specified per component, the configuration must ensure that all recursively required instances are available.

Resource constraints can be modeled in various ways. For the sake of simplicity, this article relies on a simple but powerful model that is also used in [21]. Each instance specifies its local resource requirements in advance as an integer vector where each dimension denotes a specific resource and the corresponding value denotes the required amount. The vector might vary depending on the usage of the instance. Similarly, the available resources on each device can be modeled as

a vector. Since the availability of resources can change, the values might fluctuate at runtime. To satisfy the resource constraints, the configuration has to ensure that at any time the index-wise sum of all requirement vectors of local instances is index-wise less than or equal to the vector that specifies the locally available resources.

The complexity of automatic configuration arises from the fact that both, resource and structural constraints must be fulfilled simultaneously. Due to the recursive definition of structural constraints in PCOM, it is not possible to calculate the resource requirements of a certain subtree in advance without determining all possible configurations of that subtree. But even if it was possible, the strictly limited resource availability might lead to exclusions between structurally possible configurations of arbitrary subtrees. Note that in general finding all exclusions is as complex as finding a configuration.

A. Example

In the following, we briefly describe the process of automatic configuration based on PCOM using an exemplary application. Figure 1 depicts an environment that consists of three devices. Each device has a certain amount of resources. The PDA has a single display (*DSP*), a certain amount of memory (*MEM*) and *CPU*. Each device hosts some components. The laptop hosts a component that enables a remote system to access the file system (*File System*) and another one that is capable of displaying a presentation (*Remote PPT*). Each instance of this component requires *CPU*, memory and access to the local presentation library (*PLIB*). Furthermore, an instance of this component requires two displays.

If an instance of the application anchor (*PPT Control*) is started, the container on the PDA must assign the resources and it must resolve the dependencies (*Input* and *Output*). To resolve the dependencies, it has to query the containers that are currently available in the environment for components that are capable of creating compatible provisions. Thereafter, it can decide to use one of the possible components to satisfy the dependency. To satisfy the resource requirements the container must assign local resources that match the requirements of the component. In this example, *Input* can be resolved using an instance of the *File System* component on the laptop or on the desktop. *Output* can be resolved by *Remote PPT* on the laptop or *Local PPT* on the desktop. If the PDA uses the *Remote PPT*, the laptop must assign the resources and resolve the displays. In this environment, there are four structural possibilities to configure the application depending on the choice for *Input* and *Output* (see Fig. 2 and 3). Since the *Imager* can only be started once due to the limitation of display resources, there is no way of using *Remote PPT* component in such a way that all constraints are met (see Fig. 3). The two executable configurations use a *File System* on the desktop or on the laptop and the *Local PPT* (see Fig. 2). This example also demonstrates the interrelation of resource and structural constraints. Choosing an instance that represents a locally valid option can still lead to unsatisfiable requirements that can only be discovered gradually.

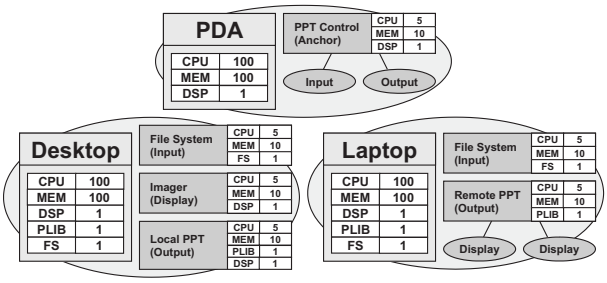


Fig. 1. Exemplary environment.

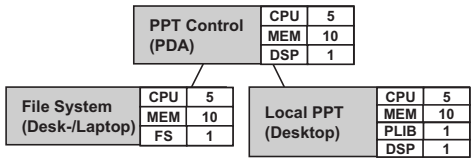


Fig. 2. Executable configurations.

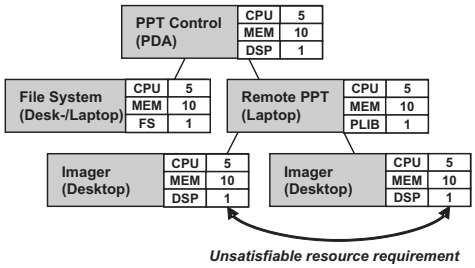


Fig. 3. Not executable configurations.

B. Requirements

The requirements towards peer-based automatic configuration can be derived directly from the presented system model and the overall vision of Pervasive Computing with respect to the distraction-free support of user tasks:

- *Completeness:* If a valid configuration exists automatic configuration should be able to determine one. Also, it should be capable of detecting that a certain application is currently not executable at all. Otherwise, users might eventually become frustrated. However, since the problem of finding a single configuration is NP-complete, achieving completeness for arbitrary problem instances is not practicable. Thus, in practice we can only demand that automatic configuration is capable of finding solutions in a broad range of different environments. As we will discuss in the evaluation section, a complete approach whose runtime is limited is often preferable over a heuristic that ”arbitrarily” ignores a number of possible solutions.
- *Efficiency:* As the configuration delay of complete solutions for automatic configuration will increase exponentially with the size of the problem, efficiency becomes a major requirement. Since long configuration delays might lead to frustrated users, automatic configuration should include as many optimizations as possible to enable speed-ups without overloading resources or sacrificing completeness.

- *Optimism:* Ideally, an algorithm for automatic configuration should be fast in resource-poor as well as resource-rich environments. Typically there is a trade-off between optimizing worst- and best-case scenarios. Since users would expect to achieve speedups by adding resources, optimizations of the worst-case delays at the cost of higher execution times in resource-rich environments are not desirable. Therefore, automatic configuration should be optimistic.
- *Distribution:* In peer-based systems the availability of a powerful and reliable device cannot be guaranteed. As a result, the scalability of a centralized approach will be limited in environments that consist of a large number of resource-poor devices. In order to utilize the inherent parallelism and the resources of such environments, automatic configuration should be performed cooperatively by the available devices.
- *Resilience:* The mobility of users and devices in pervasive systems leads to continuous and possibly unpredictable fluctuations regarding the availability of functionalities. As a result, applications in such systems have to cope with the resulting dynamics at runtime. Since an algorithm for automatic configuration might be running a couple of seconds, the algorithm itself should be capable of dealing with fluctuations that can be detected during its execution.

IV. Approach

In general, finding a single executable configuration in the presence of strictly limited resources is an NP-complete problem. This can be shown, for instance by interpreting a conjunctive normal form that is known to be NP-complete for more than three literals [6] as components with specifically constructed resource constraints. As a result, approaches for automatic configuration can apply NP-complete formalisms.

As we will show in the following, automatic configuration can be mapped to a Constraint Satisfaction Problem. Informally, Constraint Satisfaction Problems can be described as follows: Given a set of variables with finite domains and a set of constraints between variables, find a valid variable assignment such that all constraints between the variables are met.

Due to the specifics of the peer-based system model, centralized approaches towards solving Constraint Satisfaction Problems cannot fulfill the requirement regarding distribution. The foundations for distributed algorithms have been developed in the field of Distributed Artificial Intelligence. In this field, the notion of Distributed Constraint Satisfaction Problems has been formalized [22]. There, the set of variables and constraints is distributed across a number of agents. Each agent is responsible for assigning its variables and evaluating its constraints. An overview and a classification of distributed algorithms for solving such problems can be found in [23].

From this set of algorithms, we show how Asynchronous Backtracking [22] can be extended to fulfill all requirements towards peer-based automatic configuration. Asynchronous Backtracking is a sound and complete dependency-directed backtracking algorithm. It enables agents to concurrently assign values to their variables and thus has the potential to use the available parallelism. As we will show later on, it is possible to construct a mapping that enables the algorithm to start processing without any further distribution of knowledge. In the best case, it simply assigns all variables the right value and terminates. In contrast to consistency algorithms that first try to eliminate some illegal options before they assign values to variables, this algorithm fulfils the requirement towards optimism. Apart from that, the dependency-direction of the algorithm reduces the search within irrelevant possibilities (thrashing) by only considering options during backtracking that have the potential to resolve the conflict. This has the potential to greatly increase the efficiency of automatic configuration in many environments. Finally, as we will discuss later on, an extension to achieve resilience can be added in a straight-forward manor.

A. Configuration as Constraint Satisfaction

To use existing techniques for solving Constraint Satsifaction Problems as basis for automatic configuration, the functionalities present in an environment as well as structural and resource constraints must be represented as variables, domains and constraints between variables. To model PCOM applications, we map dependencies to variables, components used to satisfy a certain dependency to their domains, and the application structure with resource requirements to constraints.

To model structural constraints, each component instance is represented as a multi-dimensional variable where each dimension denotes one of its dependency. The domain of each dimension is given by the available options for the corresponding dependency. For the *PPT Control* (see Fig. 1) that has two dependencies *Input* and *Output*, we create a two dimensional variable. If there are two possibilities to satisfy the dependency *Output* (*Remote PPT* and *Local PPT*) and one for the dependency *Input* (*File System*), the domain of the variable will be $[0, 1], [0]$. Note that the domain solely contains direct possibilities. Due to the recursive nature of dependencies, there might be many possibilities to configure each of the assignments, e.g. if there were multiple *Imagers*, there would be multiple ways to configure a *Remote PPT*.

A fundamental difference between constraint satisfaction and automatic configuration is the fact that constraint satisfaction determines an assignment for all variables. Automatic configuration only needs to determine a partial solution that satisfies the constraints, i.e. if an instance is not used within the configuration, the dependencies of this instance must not be resolved. To model this, the domain of each dimension is extended with the pseudo value -1. Thus, the domain for the previous example would be $[-1, 0, 1], [-1, 0]$. One can think of the dependencies whose component has not been discovered and used as set to -1. This effectively transforms the search for a partial solution in a search for a complete solution.

Now that the variables and domains are defined, the mapping must ensure that only structurally valid configurations are generated. This can be achieved by two constraints. Both can be motivated by looking at the *Output* dependency of

the previously introduced example. There are two possible instances (*Remote PPT* and *Local PPT*) that can be selected to fulfill this dependency. Since the configuration requires only one at a time, we can add the constraints that *Remote PPT* needs to be considered iff its parent instance assigns values that contain 0 in the first dimension. Similarly, *Local PPT* needs to be considered iff 1 is assigned to the first dimension of the variable (see Equation 1). Furthermore, another constraint is required that ensures that the pseudo value is used iff the component instance is not used by its parent (see Equation 2). Apart from these recursively defined constraints, one additional constraint must be used to ensure that the anchor is always instantiated. Otherwise, the trivial configuration that contains only a non-instantiated anchor also fulfils all structural requirements (see Equation 3). Finally, the configuration must consider the resource constraints on each container. Thus, we add a constraint to ensure that the resource consumption of all instances that are executed on the container must not exceed its available resources (see Equation 4).

More formally, we can model these constraints using the following definitions and equations. Let $Asg(c,n)$ be defined as the current assignment for the variable dimension n of component instance c. Each non-anchor instance c with m dependencies that is referenced by dependency j under the value assignment k of its parent instance p is subject to:

$$\text{If } Asg(p,j) = k: \ \forall n \in \{1,\ldots,m\} \ Asg(c,n) \neq -1 \quad (1)$$

$$\text{If } Asg(p,j) \neq k: \ \forall n \in \{1,\ldots,m\} \ Asg(c,n) = -1 \quad (2)$$

Each anchor instance a with m dependencies is subject to:

$$\forall n \in \{1,\ldots,m\} : Asg(a,n) \neq -1 \quad (3)$$

Each container that has the resources $r_1,\ldots,r_m$ and hosts the instances $c_1,\ldots,c_n$ that require the resources $r_{c_i,1},\ldots,r_{c_i,m}$ is subject to:

$$\begin{pmatrix} 1 \\ \ldots \\ r_m \end{pmatrix} \geq \sum_{i=1}^{n} \begin{pmatrix} c_i,1 \\ \ldots \\ r_{c_i,m} \end{pmatrix} \quad (4)$$

B. Asynchronous Backtracking

Before we discuss further considerations that are specific for Asynchronous Backtracking, we will briefly outline the basic algorithm. For the sake of brevity, we will only provide the general idea. A detailed description including pseudo code and examples can be found in [22].

The basic version of Asynchronous Backtracking assumes that each agent is responsible for exactly one variable of the CSP. Furthermore, in order to guarantee termination, it requires a total priority ordering between variables. Each pair of variables that shares an initial contraint is connected via a directed link from the agent with the higher priority to the agent with the lower priority. The agents with higher priority variables send their current assignment to linked agents with lower priority variables. The lower priority agents, in turn, evaluate the constraints that they share with higher priority agents. Whenever an agent receives an assignment it choses its own value in such a way that it does not conflict with its constraints and then it sends its own assignment to all linked agents.

If an agent detects that it cannot assign a value to its variable in such a way that does not conflict with a constraint, it creates a new constraint. This constraint contains the set of assignments from higher priority agents that cause the conflict. The agents picks the assignment with the lowest priority and sends the conflict to the agent that created this assignment. If this agent receives the conflict it checks whether the conflict is still valid (by checking whether the assignments contained in the new constraint correspond to its own personal knowledge of assignments in CSP). If it is not valid (if it knows that some agent contained in the constraint has already changed its assignment to a different value) it simply ignores the constraint and sends its current assignment to its linked agents. If it is valid, it establishes new links from all agents to itself that participate in the constraint and that have not been linked so far. Then it records the constraint as new constraint and it tries to assign a new value to its variable that adheres to all locally known constraints.

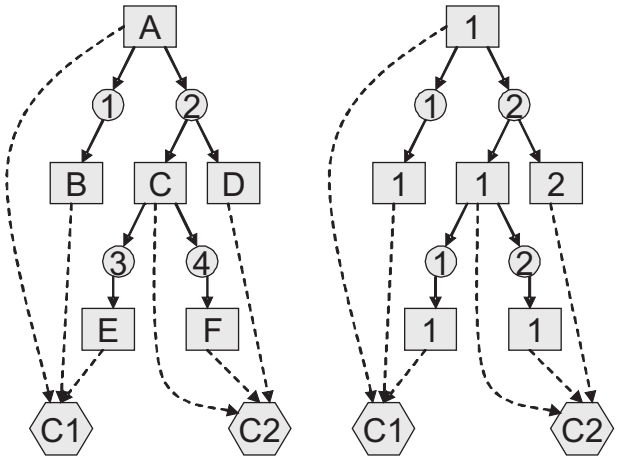
Note that since the agent that detects a new constraint will send this constraint to the agent with the lowest priority, new links will always be created from higher priority agents to lower priority agents. This, together with the fact that the inital links also pointed from higher priority agents to lower priority agents, ensures that there will be no cycles.

The algorithm starts in parallel by letting each agent assign some value to its variable in such a way that it does not conflict with its known constraints. The algorithm terminates unsuccessfully when some agent cannot assign a valid value to its variable and there are no more higher priority agents that could change their assignments in order to resolve the conflict. If a valid assignment is found, the algorithm will simply stop creating new messages.

C. Extensions for Asynchronous Backtracking

To guarantee termination, Asynchronous Backtracking requires a total priority ordering between variables to create a cycle-free constraint network. Note that this ordering also defines the strategy for resolving conflicts during backtracking. A partial ordering is introduced by the structural constraints of applications, i.e. each child instance must have a lower priority than its parent. The remaining degree of freedom can be filled by some arbitrary ordering scheme. However, in order to be usable, Asynchronous Backtracking must be able to create the scheme gradually. Otherwise the search space would have to be unfolded upfront which conflicts with the requirement of optimism.

To demonstrate this, consider the search space shown in Fig. 4(a) that consists of components *A*-*F*, with the dependencies *1*-*4*, and containers *C1* and *C2* where *A*, *B*, *E* reside on container *C1* and *C*, *D*, *F* reside on container *C2*. The dashed lines indicate resource constraints and the solid lines indicate structural constraints. Note that the structural exclusion constraint between *C* and *D* is implicit since *A* will only assign one value at a time for its dependency *2*. Also note that the dependencies and the corresponding component instances can be thought of as variables and domains of the CSP.



(a) Environment (b) Numbering

Fig. 4. Numbering Scheme.

Since *A* is the only component that is known a priori and the others must be discoverable in parallel, we can only introduce a local ordering as shown in Fig. 4(b) by assigning locally unique ids to dependencies and their possible options. From these local ids, we can create a global id by concatenating the ids along the path (e.g. $1, 2, 1, 1, 1$ for *E* or $1, 2, 2$ for *D*). On these ids, we can now define comparison operators to establish a total ordering. In order to adhere to the partial ordering introduced by the structural constraints, we must ensure that all ids that are totally included as a prefix in another id have higher priority, e.g. $1, 2, 1, 1, 1 < 1, 2, 1$. Apart from that we can for instance either decide that the length of an identifier is first compared and the longer the identifier, the lower the priority and for identifiers with equal length, we use their values for comparison. Another possible option would be to compare the values first before comparing the length.

That way an ordering can be established that would lead to a backtracking strategy where lower levels would be reconfigured before higher levels are. Alternatively one could define an ordering where backtracking would take place in one subtree before it moves to the conflicting component of another subtree.

Figure 5 shows such orderings. Note that in order for this ordering to work, every component needs to know its place within the tree, i.e., the concatenated id. Thus upon its first usage, each component needs to be supplied with the identifier that its parent assigns for it. The parent can easily create this id locally by concatenating its local id with a unique id for the dependency and the value of that dimension under which the specific child component is selected.

For our implementation we have chosen the strategy that reconfigures components on lower levels first. The idea is that lower level components have less recursively required instances that must be notified if their parent changes and thus, reduces the communication overhead. However, this is a heuristic and there are cases where this leads to higher overhead.

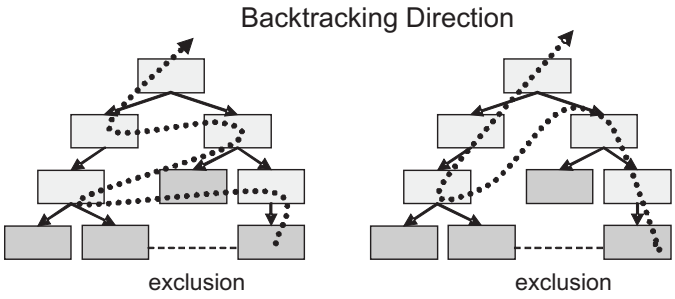


Fig. 5. Traversal Strategies.

As stated in Section III-B, automatic configuration must be capable of dealing with fluctuations that occur during its execution. Such fluctuations might be the result of the unavailability of local resources or remote devices. In pervasive systems, these fluctuations are typically hard to predict. Consider for instance a user that removes a USB device from a laptop or a traveling user that carries a number of devices. Fortunately, failure-handling for both types of fluctuations can be added in a relatively straight-forward way. Whenever the unavailability of a device is detected, the algorithm on every remaining device simply removes all assignments that have been created by this device and creates an additional constraint for every instance that has been used on the unavailable device. These new constraints state that these instances can never be used. Since Asynchronous Backtracking does not impose any timing constraints on the reception of constraints, they can be added without further precautions. Similarly, if a required resource becomes unavailable, the corresponding constraints must be added. The new constraints will eventually lead to a reconfiguration or an unsuccessful termination of the algorithm.

Asynchronous Backtracking terminates unsuccessfully if an empty constraint set is generated during the execution, i.e. if there is no further choice that can be reconsidered in order to resolve an unsatisfied constraint. Due to the tree structure of our application model, such an empty set can only be generated by the anchor. All other component instances can always ask their parents to reconfigure themselves in such a way that they are no longer used. Thus, an unsuccessful run will be recognized by the anchor. The successful termination of the algorithm is achieved if all participating devices stopped generating new messages and all messages have been delivered and processed. Therefore, detecting the successful termination is an instance of a Distributed Termination Problem.

As the termination protocol must be resilient to mobility, a simple protocol as described in [8] is not enough. To solve this problem, our current implementation uses a sligthly extended version of the credit-based termination protocol described in [13]. To deal with lost credits, we stop and restart the configuration process whenever a device is no longer available. However, as we do not want to lose the (possibly expensive) intermediate results that have been computed by the Asynchronous Backtracking algorithm so far, we only unset the current assignments of all components and we do not remove the constraints between them that have been discovered so far.

To reset the configuration process, the termination protocol attaches an epoch value to all messages that is incremented whenever a device is removed. The configuration algorithm then ensures that only those messages with the current epoch value are processed.

Clearly, due to the unpredictable nature of pervasive systems, no termination protocol can guarantee that a successful termination of the algorithm will allow a successful application start up. If a resource becomes unavailable at exactly the same instant of time when the successful termination is detected, there is nothing that can be done. At the present time, the only approach that we can propose is to start the configuration process all over again if such a situation occurs. Another possibility is to start the partial configuration and determine possible adaptations. This, however, is subject of our current research and a discussion lies beyond the scope of this article.

V. Algorithm

In the following, we provide an overview of the resulting algorithm and we describe some interesting details of our implementation. For the sake of clarity, we begin with a simplified description that omits the distributed termination protocol and our extensions to achieve resilience as these aspects are orthogonal to the basic algorithm. Thereafter, we provide an example to clarify the overall process. Finally, we outline our current approach used to detect termination and our extensions for resilience.

A. Simplified Description

In this article, the algorithm is modeled as a reactive process that responds to incoming messages (*receive_* procedures*) and the pseudo code of this simplified version (see Fig. 6, 7) does not consider different applications. It should be clear that multiple applications can be supported by transfering an additional identifier as part of each message that is used to map the message to a certain application. Also, the code does not contain all optimizations, e.g. only sending messages to containers that require it. The layout borrows from the description of Asynchronous Backtracking [22]. Note that we need to extend the algorithm with the capability of hosting multiple instances and the resource validation procedure (see Fig. 7). Furthermore, in order to support the dynamic discovery of components, we add a method that performs discovery and initializes the variables on demand.

As in the original version of Asynchronous Backtracking, the algorithm uses 3 message types namely update messages, backtrack messages and link messages. Update messages send the value assignment of a parent component to a child component or to another component with a lower priority that has created a link in response to a conflict. Backtrack messages report a conflict from a component with a lower priority to the component with the lowest priority in the conflict set. Finally, link messages inform some component that it must send future value assignments to the component that requested the link. The receive method for the link message is not shown in Fig. 6 as it solely adds the link information to the corresponding component.

```
receive_update(identifier, component, value)
 // config denotes the local knowledge about an instance
 config = getConfig(identifier)
 // this happens if the instance is selected for the first time
 if (! config exists)
   // here the variables and their domains are determined
   config = createConfig(identifier,component )
 // this adds the variable assignment to the local knowledge
 config.add(value)
 // finally, all consistency checks are performed
 check_constraints(config)

receive_backtrack(identifier, conflicts)
 // determine whether the conflicts are still conflicting
 if (! conflicts outdated)
   // retrieve the addressed conflict
   config = getConfig(identifier)
   // add the conflicts as a new constraint
   config.addConstraint(conflicts)
   for each id in conflicts
     if (! connected id)
       // create links to keep informed about changed values
       create link between parent of id and config
       // add the value of the conflict to the local knowledge
       config.add(identifier)
     // temporarily copy the currently selected components
     copy = config.getAssignment()
     // perform the consistency checks
     check_constraints(config)
     if (copy == config.getAssignment())
       // if the values have been consistent also send updates
       send_update(identifier, copy) across links

check_constraints(config)
 // if there are unmatched constraints
 if (! config.isConsistent())
   // determine whether a valid assignment can be found
   if (! config.assignConsistent())
     // if not, start or continue backtracking
     backtrack(config)
   else if (reserve_resources(config))
     // else determine whether local resource constraints are met
     assignment = config.getAssignment()
     // if they are met, send the updated assignment
     send_update(identifier, assignment) across links

backtrack(config)
 if (config.isAnchor())
   terminate unsuccessfully
 else
   // determine conflicting instances sets
   conflict_sets = minimum conflict sets
   for each conflicts in conflict_sets
     // send a backtrack message to the lowest instance
     id = minimum identifier in conflicts
     // this message could be remote or local
     send_backtrack(id, conflicts)
     // remove the conflicting assignment
     config.remove(id)
   check_constraints(config)
```

Fig. 6. Basic message handling (without termination detection).

For each component that has been used during the configuration process, the container maintains a configuration object that represents the local knowledge about this component. This object stores the value assignments that have been received from other components through update messages. Furthermore, it stores the conflicts that have been received through backtrack messages. Finally, it stores and manages the variables of the component. This includes the information about links that have been created dynamically during the execution of the algorithm. As discussed in Section IV-A, the variables represent the dependencies of the component.

```
reserve_resources(config)
 // if the instance is selected by its parent
 if (config.isInstantiated())
  // and the resources are not reserved
  if (! config.isReserved())
   // try to reserve the required resources
   if (reserve resources for config)
    config.setReserved(true)
    // if the reservation succeeds, continue
    return true
   else
    // if the reservation fails determine conflict sets
    conflict_sets = minimum conflict sets
    // a flag that indicates whether the conflict has been r esolved
    reservable = true
    for each conflicts in conflict_sets
     // pick the instance with the lowest identifier
     id = minimum identifier in conflicts
     // backtrack to the parent of the lowest instance
     send backtrack to parent of id with conflicts
     // deactivate the instance that caused backtracking
     c = getConfig(id)
     c.remove(id)
     check_constraints(c)
     // determine whether the current instance is a conflic t cause
     if (config.getIdentifier() == id)
      // if it is, it will be uninstanciated after the backtracking
      reservable = false
    if (reservable)
     // the cause of all conflicts has been removed
     reserve resources for config
     config.setReserved(true)
     return true
    else
     // the instance has already been deactivated
     return false
 else
  // if the instance is not used by the parent
  if (config.isReserved())
   // remove the resource reservation
   remove reservation for config
   config.setReserved(false)
  return true
```

Fig. 7. Resource reservation and conflict detection.

Their corresponding dimensions are determined at runtime by querying the containers of the environment for components that can be used to satisfy the dependency.

Clearly, in an actual implementation the configuration object also has to store information about the devices that host the components and factories that provide them. Furthermore, the update messages that are used to initalize a variable must contain additional information, i.e. the contract of the component and the device that uses the component. Similarly, the backtracking messages must contain the device identifiers of the variable assignments contained in the conflict set as the device that receives the message might not know on which device the corresponding variable resides.

Since each container can host multiple component instances, the algorithm must be capable of uniquely identifying them. To globally identify an instance and its position within the application, our implementation uses the generated ID discussed in Section IV-C. The application anchor has the ID $\{\}$, the first instance for the first dependency of the anchor is identified by $\{(0)[0]\}$. The second instance for this dependency is identified by $\{(0)[1]\}$. Thus, IDs are arbitrarily long sequences of pairs, where the first index of a pair denotes the dependency and the second index denotes the instance used to satisfy this dependency.

Whenever a container receives an update message, it must first retrieve the corresponding configuration object. Thereafter, it performs basically the same steps as in the original version of Asynchronous Backtracking. However, as the configuration process must take care of structural as well as resource constraints, the container must ensure that apart from the built-in structural constraints (see Equation 1, 2, and 3 in Section IV-A) all resource constraints are met (see Equation 4 in Section IV-A). To do this, it uses the resource reservation procedure shown in Fig. 7. This procedure reserves or releases the local resources that are required for a certain component depending on the value assignment of its parent. If the parent of the component assigns a different value than the value that corresponds to the component, the component is not selected and thus, its resources must be released. Otherwise, i.e. if the componet is selected by its parent, the corresponding resources must be reserved. If the resource reservation fails, the container must create and issue a corresponding backtracking message that contains the conflicting set of components.

The steps that need to be taken whenever a backtrack message is received are identical to Asynchronous Backtracking (see Section IV-B). If at some point a backtrack message would contain an empty conflict set, the algorithm would simply terminate unsuccessfully. Note that due to the tree-structure of PCOM applications this can only happen at the application anchor as other components can always add the value assignment of their parent under which they are selected to the conflict set.

Since this algorithm is essentially an instance of Asynchronous Backtracking, the proof of correctness follows the argumentation provided in [22]. The algorithm will not stop sending messages until a valid configuration has been found or an empty conflict set has been created. Due to the total ordering of variables and the invariant that the dynamically created links will not introduce cycles, the algorithm will terminate eventually.

B. Examplary Configuration

To describe the configuration process performed by the algorithm in a more dynamic manner, we will use the presentation application example introduced earlier (see Fig. 8). When the user starts the application, the container calls the *receive_update* procedure with the ID $\{\}$, an identifier that locally identifies the *PPT Control* component and the value $\{\}$. This signals that an anchor should be started (a). Since this is the first time that the configuration algorithm sees an update for $\{\}$, it creates a configuration object for this instance. Using the contract of the instance, it determines that *PPT Control* has two dependencies, thus it creates a two-dimensional variable $[Input]$,$[Output]$. To determine the domain of the variable, i.e. possible options to satisfy the dependencies, the container performs local and remote lookups (b). Thereby, the algorithm discovers the following options: *File System (desktop)* $\{(0)[0]\}$, *File System (laptop)* $\{(0)[1]\}$, *Remote PPT (laptop)* $\{(1)[0]\}$ and *Local PPT (desktop)* $\{(1)[1]\}$. Thus, the domain is $[-1, 0, 1]$,$[-1, 0, 1]$. For the new variable, the initial assignment is $[-1]$,$[-1]$.

The algorithm continues to add the value $\{\}$ to the local knowledge which states that the instance bound to the configu-

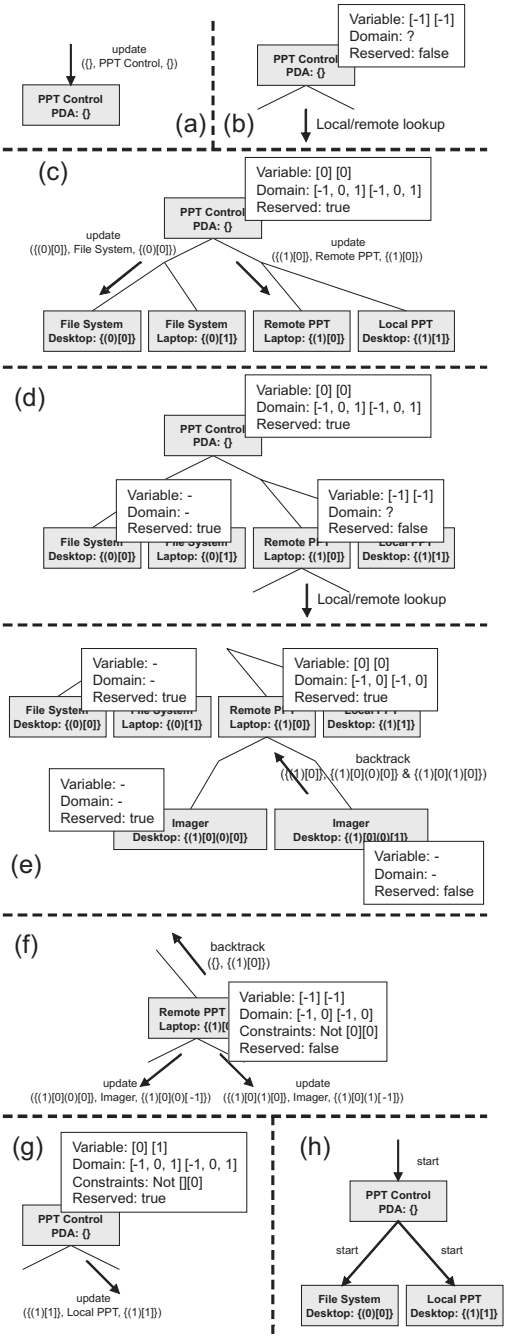


Fig. 8. Configuration Process.

ration object is instantiated. Thereafter, the algorithm calls the *check_constraints* procedure and determines that the current assignment $[-1]$,$[-1]$ is not valid, since the instance is used according to the local knowledge. Note that this is a result of the built-in constraints presented in Section IV-A. Next, the algorithm determines a valid assignment $[0]$,$[0]$ and reserves the resources using the *reserve_resources* procedure. The reservation finishes successfully and the algorithm continues to send parallel update messages to the *File System* $\{(0)[0]\}$ and the *Remote PPT* $\{(1)[0]\}$ (c).

When the update message for the *File System* arrives, the algorithm creates the configuration object, adds the value to the local knowledge, performs the resource reservation, and stops without sending further messages (d). In response to the update for the *Remote PPT*, the algorithm sends two updates to the *Imager* ($\{(1)[0](0)[0]\}$ and $\{(1)[0](1)[0]\}$). The first update message creates a new configuration object and finishes successfully. The second update fails due to a lack of resources. Thus, the *reserve_resources* procedure determines that the minimum conflicting sets consist of exactly one set of component instances that contains both instances of the *Imager* component (e).

Note that although the *File System* is also running on the desktop, its identifier will not be added to the conflict set since it has nothing to do with the shortage on displays. Furthermore, the algorithm does not need to add the complete path to the anchor to the constraint as it can be gradually generated whenever a conflict is escalated. Following the traversal strategy, $\{(1)[0](1)[0]\}$ is picked as the smallest identifier and a backtrack message is sent to is parent. Additionally, the instance is deactivated and all potentially reserved resources and required instances are released by calling *check_constraints*.

When the backtracking message arrives at the *Remote PPT*, the component will determine whether it has to create any new links. Since both identifiers contained in the conflict set are local variables, no new link must be created. Therefore, the algorithm continues to add a mutual exclusion constraint between $\{(1)[0](0)[0]\}$ and $\{(1)[0](1)[0]\}$ to the local knowledge. In cases where added conflicts are not conflicts between linked instances, the addition of new links between the assigning instance and the instance that recorded the constraint are necessary to ensure that the constraint evaluation always considers all relevant variable assignments of the present situation.

Since the *Remote PPT* cannot create a valid assignment, it creates a backtracking message that contains its own identifier and sends it to its parent. Thereafter, the *Remote PPT* is deactivated and its constraints are checked again. Thereby, the algorithm releases all resources, assigns $[-1]$,$[-1]$ and creates updates that will eventually release previously bound instances (f). When the *PPT Control* receives the backtracking message, it adds the constraint that the *Remote PPT* can never be started and assigns another value for the *Output* dependency. It selects the *Local PPT* $\{(1)[1]\}$ and it creates an update (g). When the update arrives, the *Local PPT* will be reserved and the algorithm stops.

When the algorithm succeeds, the application must still be started. Therefore, an asynchronous traversal of the tree-structure starting from the application anchor is sufficient. This will not result in conflicts, since each configuration object has already reserved the resources for the chosen bindings (h). After the application has been started, all configuration objects that have been created can be removed.

C. Termination and Resilience

To detect the fact that the configuration process has finished successfully, a distributed termination detection protocol is required. Such a protocol can be added by wrapping the *receive_** procedures and the send statements. The necessary

steps that need to be performed depend on the chosen protocol (see [8], [12], or [13] for details).

For our current implementation, we are using an extended version of the credit-based protocol described in [13]. As this protocol is not resilient by itself, we restart the termination detection process whenever a device becomes unavailable. To do this, we identify the current termination detection phase by an epoch counter that is monotonically increased upon failures.

At the beginning of the configuration process the device that hosts the application anchor determines the set of available devices and sets its current epoch to zero. Thereafter, it sends the set of devices and the epoch to all devices contained in the set. Whenever a container sends a message, it piggybacks a credit and its current epoch. If a container receives and processes a message, it uses the received credit to derive new credits for messages that are generated by processing the original message. If a container determines that it has terminated locally, i.e. if it has processed all messages, it sends the remaining credits to the container that hosts the anchor. If the anchor detects that it has received all credits for the current epoch, the whole configuration process has terminated successfully.

Whenever a device becomes unavailable during the process, some container will detect this, for instance by receiving an exception when it tries to transfer some message. At that point, the container sends a message to the container of the anchor. This message contains the device that is no longer available. Upon receiving this message, the anchor container determines whether the removed device is part of the set of devices. If it is part, it removes the device and increases the epoch counter and transfers a message to all remaining devices that contains the new epoch and the device has been removed. If the device is not part of the set of devices, it has been removed already and thus, the message can be ignored.

When the other containers receive the removal message, they drop all messages that are not part of the contained epoch and add additional constraints. If a component has some dependency that can be fulfilled by a component that is hosted by the removed device, they add a contraint that contains solely the identifier of this component. Thereafter, they remove all links that point to the removed device and set all variable assignments to -1. Thereby, they release all resource reservations and remove all assignments that they have received so far. This will bring the containers into a consistent state. This state is identical to the initial state with the exception that the constraints that have been discovered so far will not be lost.

After all containers have adjusted their states the anchor starts the configuraton process again by recomputing its assignments and sending the assignments to all linked containers. Thereby, the anchor will add credits to the messages that have been created from a new inital credit and it will add the new, increased epoch. All agents will keep dropping all messages that they receive from other agents that contain an epoch with a lower value than their own.

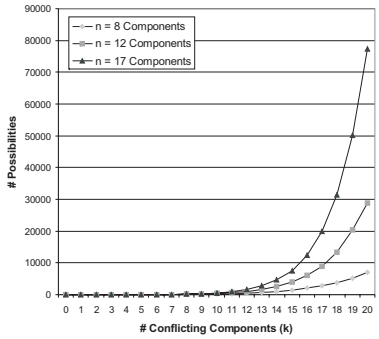


Fig. 9. Average number of structural possibilities.

VI. Evaluation

As discussed in Section IV, Asynchronous Backtracking fulfills the requirements regarding completeness, optimism and distribution. Furthermore, it can be extended to fulfill the requirements with respect to resilience. In this section, we discuss efficiency as the last remaining requirement. To do this, we first discuss parameters that have an impact on the configuration delay. Thereafter, we describe the setup for our experiments and we present and discuss the results of a number of simulations and real-world measurements.

A. Discussion

Asynchronous Backtracking resolves unrelated conflicts simultaneously and it reconsiders only those instances that have the potential to resolve a conflict. Thus, the configuration complexity depends on the induced width, i.e. the size of sub problems that can be solved independently, and not the total width of the search space [2]. The induced width of automatic configuration depends on the number of structurally valid configurations and the locality of resource conflicts, i.e. the number of instances that have conflicting requirements towards the same resources.

In many pervasive systems, resource conflicts can be assumed to be relatively local. To justify this, consider that the worst-case runtime occurs, if many instances are executed on one device and a widely used resource (e.g. memory or CPU) is not available. However, the integration of devices into everyday objects leads to environments where the majority of devices are specialized embedded systems. Just like everyday objects, they will be tailored towards a small number of specific functionalities, which will increase the locality.

The number of structurally valid configurations depends on the number of available components that can be used within the application and thus, it heavily depends on the capabilities of the environment.

B. Experimental Setup

To analyze the effects of different degrees of conflict locality and various application and environment sizes, we constructed sample environments using the following procedure:

We create an application that consists of n instances by adding n components to a binary tree from left to right, top to bottom. Then we create one container and place the anchor on

it. For the remaining (n-1) components we create m containers and place them on the containers round-robin. Thereby, we set the resource requirements of each component to one unit of one resource that is used by all components on the container. Furthermore, we set the available amount of the resource to the number of components that are hosted on this container. Then we randomly pick k components and replicate them on randomly selected containers. Hereby, we set their resource requirements for the commonly used resource on that container to two without increasing the resources on this container.

As a result, increasing k will lead to a higher potential for conflicting selections during automatic configuration and decreasing the number of containers m will decrease the locality of the resulting conflicts. Note that there will always be exactly one configuration that can be started which consists solely of instances provided by the initially placed components.

Figure 9 shows the average number of structural possibilities that result from 100 randomly created scenarios for applications with n = 8, 12, and 17 components and k = 0 to 20 duplicated components. The exponential growth in the number of structural possibilities can be attributed to the way conflicting components are introduced, i.e. by replicating existing components. If a component is duplicated and added, it can use all existing combinations of child components that the existing copies were already able to use.

C. Simulations

To analyze the effects of an increasing number of possibilities we ran a number of simulations with a discrete event simulator. Within one time step, the simulator processes all messages that have been sent and creates all new messages before it moves on to the next time step.

Figures 10, 11 and 12 depict simulation results in cases where the locality of conflicts is high, i.e. m = (n-1) / 2, for different application sizes (n = 8, 12, 17) and a different number of conflicting components (k = 0 to 20). Each measurement shows the average, respectively the maximum, of 100 runs.

The simulations show that the number of messages required to determine configurations grows exponentially (see Figs. 10 and 11). For $k \geq 17$ the maximum number of messages exceeds *400*. This is a result of the exponential increase of structural possibilities. Note that the number of messages does not necessarily lead to a high configuration delay as the solution is found in less than *90* time steps (see Fig. 12). In a real system, the exact amount of messages as well as the overall configuration delay might vary depending on the message delay. For instance, if a device requires a long time to detect or propagate a local conflict, the number of messages as well as the required duration might increase or decrease depending on the scenario.

One might argue that the worst-case message overhead prohibits the application of the algorithm. Therefore, we have compared the achievable completeness if the number of messages is limited to 100, 200, 300 and 400 with the completeness that can be gained from a greedy heuristic that selects sub trees recursively without ever reconsidering a choice as proposed in [3]. Figures 13 and 14 show the

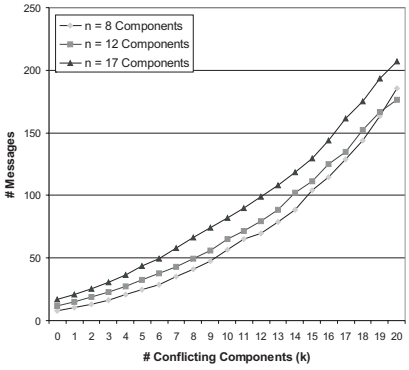


Fig. 10. Average number of messages with high locality (m = (n-1) / 2).

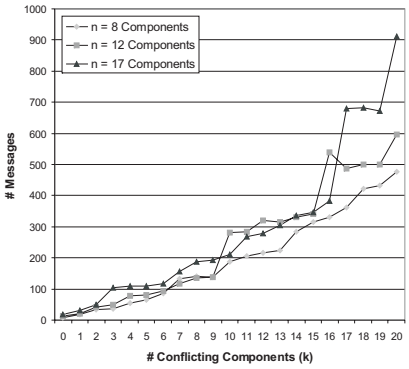


Fig. 11. Maximum number of messages with high locality (m = (n-1) / 2).

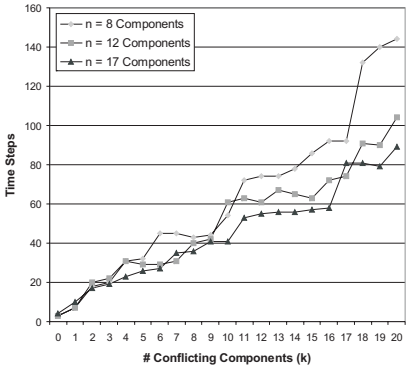


Fig. 12. Maximum duration with high locality (m = (n-1) / 2).

success rates for an application with 12 instances. In average, the heuristic produced 23-100 messages, but for k = 15, it can only find the configuration in 8 cases whereas backtracking finds 59 with 100 and all with 400 messages. Thus, even if the complete algorithm would have been manually aborted, the success rate would have been higher.

If we construct a scenario where there is no valid configuration by increasing the resource requirements of one initially placed component by one, the number of transferred messages increases by approximately a factor of two. This can be attributed to the min-conflict value ordering heuristic that is used to select instances. However, in over-constrained search spaces aborting the process does not affect completeness.

Finally, Figs. 15 and 16 show the success rate in a case where the locality assumption of conflicts does not hold.

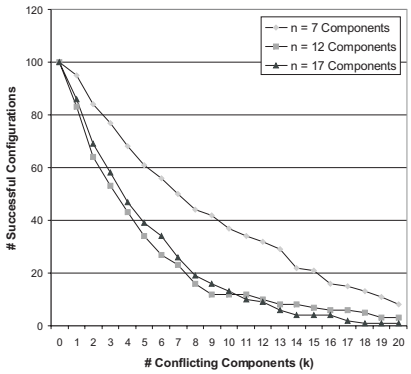


Fig. 13. Greedy completeness with high locality (m = (n-1) / 2).

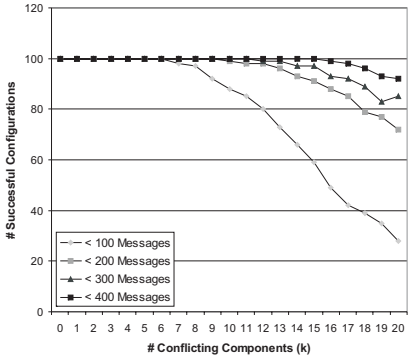


Fig. 14. Backtracking completeness with high locality (m = (n-1) / 2).

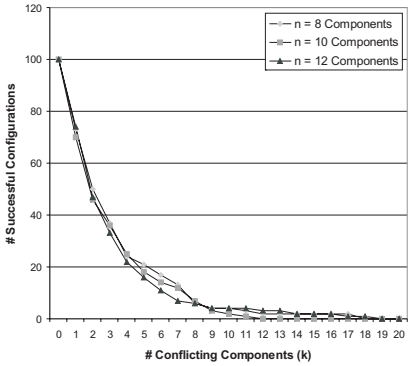


Fig. 15. Greedy completeness with low locality (m = 4).

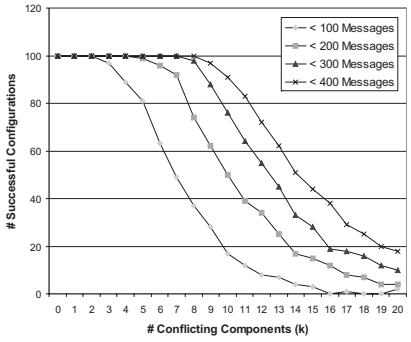


Fig. 16. Backtracking completeness with low locality (m = 4).

Instead of increasing the number of containers as the size of the application grows, we fix the number to 4. Despite the increasing message overhead, the complete algorithm is still able to outperform the greedy heuristic in terms of completeness.

D. Experiments

To determine the configuration delays in a real system, we have implemented a prototypical version of the algorithm as part of PCOM. To provide values for small devices, we placed an application with 7 components on two Pocket PCs (XScale 400MHz / 10 MBit WLAN) using the procedure described in Section VI-B. Since all components were using the same resource on each of the Pocket PCs, this experiment reflects a situation where the locality of conflicts is low.

We ran 7 scenarios with 0, 2, 4, 6, 8, 10 and 12 randomly created conflicting components. For each number of conflicting components we performed 10 randomly created runs. Figures 17, 18, and 19 show the results of these measurements with respect to configuration delay, average and maximum number of messages as well as the achievable completeness within bounded delays. The delay as well as the number of messages additionally include the overhead introduced by the distributed termination detection protocol and application startup.

Our measurements show that the completeness of the backtracking algorithm that can be achieved with bounded delay is always higher, even if the delay is limited to 10 seconds. Note that this is only slightly higher than the average runtime of the greedy algorithm which lies between 8 and 9 seconds.

VII. Related Work

As most projects in Pervasive Computing deal with distributed functionalities, they have to address the management of compositions. The degree of automation varies heavily depending on the focused system model. The GAIA project [17] for instance separates the implementation of functionalities from the composition of applications using two externalized mappings. Since GAIA assumes a partially static environment, it is not necessary to automate these mappings. The AURA project [9] uses a task abstraction that is mapped onto functionalities available in a certain environment. The mapping is done by a centralized environment manager that coordinates the functionalities of its environment. In contrast to PCOM and GAIA, functionalities in AURA are self-contained entities that solely interact implicitly through users. The iROS [15] system provides a generic mechanism that enables interaction between functionalities. Since iROS does not impose constraints on the available components, the management of the composition must be performed manually. Similarly, one.world [10] does not support the automated management of compositions. Instead, the system shifts this responsibility to the developer.

The Pebbles project [18] uses an abstraction called goal to model an application and it uses a planning engine that automates the creation of valid configurations at runtime. To the best of our knowledge Pebbles uses a centralized planning engine. Another system that uses centralized planning

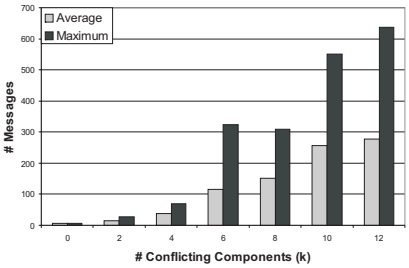


Fig. 17. Number of messages for two devices without locality (n=7).

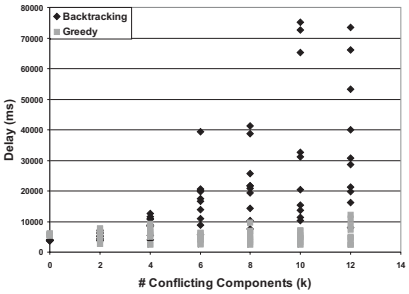


Fig. 18. Total delay for two devices without locality (n=7).

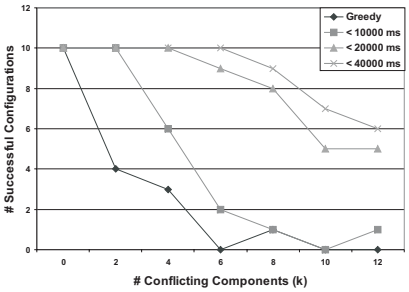


Fig. 19. Completeness for two devices without locality (n=7).

to configure component-based applications is Planit [1]. Planit uses temporal refinement planning to adapt and configure applications at runtime. In contrast to the proposed approach, centralized approaches require a global view of the components and resources of the environment.

Automatic configuration as discussed in this article can be seen as instance of a distributed resource allocation problem. In the past, research has been applied to distinct domains, e.g. job scheduling [20] or patient scheduling [7]. However, these domains are different from automatic configuration since they try to allocate a set of tasks that is known in advance. In the discussed approach the set of components is discovered at runtime. This requires that the set of variables and domains can be gradually created from the discovered components.

More recently, researchers developed the notion of Dynamic Distributed Constraint Satisfaction Problems, e.g. to perform distributed monitoring in sensor networks [14]. To deal with the dynamics of the environment, constraints need to be added or removed depending on a predicate. This is similar to the proposed extension for resilience as all constraints depend on a predicate that continuously evaluates the availability of devices and resources. However, the approach presented in [14] does not deal with discovery.

VIII. Conclusion

In this article, we have discussed the requirements on automatic configuration in peer-based pervasive systems. Furthermore, we have presented a mapping that enables the automatic configuration of component-based applications in PCOM using Distributed Constraint Satisfaction techniques. The feasibility of this approach has been evaluated using simulation and a prototypical implementation of the algorithm. The results indicate that the presented complete approach is preferable over the greedy heuristic. Although it is possible to construct scenarios in which the complete algorithm will have an unacceptable delay, we are confident that many real-world problems will exhibit the locality to keep the delay within acceptable bounds.

In the near future, we will extend the presented work towards runtime adaptation where the cost for reconfiguring an executed partial application must be taken into account. Also, we are planning to investigate hybrid systems that might contain coordinating entities at certain times. In such systems, a fragment of the state of the environment could be collected at each of the available coordinators which in turn could thereafter cooperatively configure applications.

Acknowledgments

This work is funded by the German Research Foundation (DFG) as part of the Priority Programme 1140 - Middleware for Self-organizing Infrastructures in Networked Mobile Systems.

References

[1] Naveed Arshad, Dennis Heimbigner and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *ICTAI '03: 15th IEEE International Conference on Tools with Artificial Intelligence*, pp. 39–46, 2003.

[2] A. R. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, University of Oregon, 1995.

[3] Christian Becker, Marcus Handte, Gregor Schiele and Kurt Rothermel. Pcom - a component system for pervasive computing. *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, pp. 67–76, 2004.

[4] Christian Becker, Gregor Schiele, Holger Gubbels and Kurt Rothermel. Base - a micro-broker-based middleware for pervasive computing. *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pp. 443–451, 2003.

[5] Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, New York, NY, 1971. ACM Press.

[7] K. Decker and J. Li. Coordinated hospital patient scheduling. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, p. 104, Washington, DC, 1998. IEEE Computer Society.

[8] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[9] David Garlan, Daniel P. Siewiorek, Asim Smailagic and Peter Steenkiste. Project aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[10] Robert Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[11] Xiaohui Gu, Klara Nahrstedt, Rong N. Chang and Christopher Ward. Qos-assured service composition in managed service overlay networks. *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pp. 194, Washington, DC, 2003. IEEE Computer Society.

[12] Ten-Hwang Lai and Li-Fen Wu. An (n -1)-resilient algorithm for distributed termination detection. *IEEE Transactions Parallel Distributed Systems*, 6(1):63–78, 1995.

[13] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30(4):195–200, 1989.

[14] P. J. Modi, H. Jung, M. Tambe, W.-M. Shen and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. *CP '01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pp. 685–700. Springer-Verlag, 2001.

[15] S. R. Ponnekanti, B. Johanson, E. Kiciman and A. Fox. Portability, extensibility and robustness in iros. *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 11–20. IEEE Computer Society, 2003.

[16] B. Raman and R. Katz. An architecture for highly available wide-area service composition. *Computer Communication Journal*, 26(15):1727–1740, 2003.

[17] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[18] U. Saif, H. Pham, J. M. Paluska, J. Waterman, C. Terman and S. Ward. A case for goal-oriented programming semantics. *UBISYS '03: Workshop on System Support for Ubiquitous Computing at UBICOMP '03*, pp. 1–8, 2003.

[19] G. Schiele, C. Becker and K. Rothermel. Energy-efficient cluster-based service discovery. *11th ACM SIGOPS European Workshop*, pp. 20–22, 2004.

[20] K. Sycara and J. S. Liu. Multiagent coordination in tightly coupled task scheduling. *1996 International Conference on Multi-Agent Systems*, 1996.

[21] D. Xu, K. Nahrstedt and D. Wichadakul. Qos and contention-aware multi-resource reservation. *Cluster Computing*, 4(2):95–107, 2001.

[22] M. Yokoo, E. H. Durfee, T. Ishida and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.

[23] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

Marcus Handte is a researcher of the Distributed Systems Research Group at the Institute of Parallel and Distributed Systems at the Universität Stuttgart. He received a MSc degree in computer science from the Georgia Institute of Technology in 2002 and a Diplom in computer science from the Universität Stuttgart in 2003. Currently, he is pursuing his PhD with a focus on the area of system support for pervasive applications. He is a member of the ACM and the GI.

Christian Becker received his Diplom in computer science from the Universität Kaiserslautern in 1996 and his PhD from the Universität Frankfurt in 2001. Since 2001 he is working as senior researcher and lecturer at the Institute for Parallel and Distributed Systems at the Universität Stuttgart. His research interests are middleware platforms and context-aware computing.

Kurt Rothermel is a professor in the Distributed Systems Research Group at the Institute of Parallel and Distributed Systems at the Universität Stuttgart. His research interests include performance evaluation of distributed systems, context aware and adaptive systems, and sensor networks. He received a PhD in computer science from the University of Stuttgart. He is a member of the ACM and the GI.